Breaking a Time-and-Space Barrier in Constructing Full-Text Indices[*]

Wing-Kai Hon[†] Kunihiko Sadakane[‡] Wing-Kin Sung[§]

Abstract

Suffix trees and suffix arrays are the most prominent full-text indices, and their construction algorithms are well studied. In the literature, the fastest algorithm runs in $O(n)$ time, while it requires $O(n \log n)$-bit working space, where n denotes the length of the text. On the other hand, the most space-efficient algorithm requires $O(n)$-bit working space while it runs in $O(n \log n)$ time. It was open whether these indices can be constructed in both $o(n \log n)$ time and $o(n \log n)$-bit working space.

This paper breaks the above time-and-space barrier under the unit-cost word RAM. We give an algorithm for constructing the suffix array which takes $O(n)$ time and $O(n)$-bit working space, for texts with constant-size alphabets. Note that both the time and the space bounds are optimal. For constructing the suffix tree, our algorithm requires $O(n \log^\epsilon n)$ time and $O(n)$-bit working space for any $0 < \epsilon < 1$. Apart from that, our algorithm can also be adopted to build other existing full-text indices, such as Compressed Suffix Tree, Compressed Suffix Arrays and FM-index.

We also study the general case where the size of the alphabet Σ is not constant. Our algorithm can construct a suffix array and a suffix tree using optimal $O(n \log |\Sigma|)$-bit working space while running in $O(n \log \log |\Sigma|)$ time and $O(n(\log^\epsilon n + \log |\Sigma|))$ time, respectively. These are the first algorithms that achieve $o(n \log n)$ time with optimal working space. Moreover, for the special case where $\log |\Sigma| = O((\log \log n)^{1-\epsilon})$, we can speed up our suffix array construction algorithm to the optimal $O(n)$.

1 Introduction

Due to the advance in information technology and bio-technology, the amount of text data is increasing exponentially. To assist users to locate their required information, the role of indexing data structures has become more and more important. For texts with word boundary such as English, inverted index [7] is used since it enables fast queries and is space-efficient. However, for texts without word boundary like DNA/protein sequences or Chinese/Japanese texts, inverted index is not suitable. In this case, we need full-text indices, that is, indexing data structures which make no assumption on the word boundary. Suffix trees [20] and suffix arrays [19] are two fundamental full-text indices in the

[*] Preliminary version appears in the *Proceedings of the 44th Symposium on Foundations of Computer Science*, pages 251–260, 2003.

[†] Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan. (Email: wkhon@cs.nthu.edu.tw)

[‡] Department of Computer Science and Communication Engineering, Kyushu University, Japan. (Email: sada@csce.kyushu-u.ac.jp)

[§] Department of Computer Science, School of Computing, National University of Singapore, Singapore. (Email: ksung@comp.nus.edu.sg)

literature, which find numerous applications in areas including data mining [28] and biological research [8]. For the other full-text indices, almost all of them are originated from these two data structures.

Suffix trees and suffix arrays are very useful since they allow us to perform pattern searching efficiently. Consider a text with n characters. Given the suffix tree, we can search for a pattern P within the text using $O(|P|)$ time, which is independent of the text size. For suffix array, the searching time is $O(|P| + \log n)$,[1] which is only a bit slower. One more advantage of suffix array is that even if this indexing structure is placed in external memory, it still can achieve good I/O performance for searching [3]. In spite of that, suffix trees and suffix arrays cannot be built easily when n is large. The construction algorithms for both of them are either too slow, or require too much working space.

For instance, when we optimize the construction time, based on the work from Weiner [30], McCreight [20], Ukkonen [29], and Farach [4], a suffix tree and a suffix array can be built in $O(n)$ time. However, the working space required is $\Omega(n \log n)$ bits.

On the other hand, when we optimize the construction working space, based on the recent work by Lam et al. [18], we can first build the Compressed Suffix Array (CSA) of [6] and then convert it to the suffix tree and the suffix array. Although such approach reduces the working space to $O(n)$ bits, the execution time is increased to $O(n \log n)$. Another solution is to rely on external memory [3] to control the working space. However, the time complexity is even worse, not to mention the increase in I/O burden.

It was open whether the suffix tree and the suffix array can be constructed in $o(n \log n)$ time and $o(n \log n)$-bit working space. The need to break this time-and-space barrier is illustrated in a concrete example that arises in practice. Suppose we would like to construct a suffix array for human genome (of length approximately 3 billion). The fastest known algorithm runs in linear time. However, it requires 40 Gigabytes working space [17]. Such memory requirement far exceeds the capacity of ordinary computers. On the other hand, if we apply the most space-efficient algorithm, the working space required is roughly 3 Gigabytes, which is possible to be implemented on a PC nowadays. The time required, however, is more than 20 hours [18], which is a bit slow.

Apart from suffix trees and suffix arrays, we observe that the other full-text indices also suffer from the same time-and-space barrier during the construction phase. Such barrier may prevent these indices to become useful for large-scale applications.[2] Table 1 summarizes the performance of the best known algorithms for constructing these full-text indices.

1.1 Our Results

Our results are based on the following model. Firstly, we assume a unit-cost RAM with word size of $O(\log U)$ bits, where $n \leq U$, in which standard arithmetic and bitwise boolean operations on word-sized operands can be performed in constant time [1, 9]. Secondly, to compare our work fairly with the other main-memory algorithms, we add the following assumptions: (1) We restrict our algorithms to be running within the main memory, in which no I/O operations are involved in the intermediate steps; (2) for counting the working space, we do not include the space for the output of the full-text indices (This can be justified as output can be written directly to the secondary storage upon completion without occupying the main memory). Under the above model, this paper proposes the following construction algorithms for full-text indices, where the input text is assumed to be over a constant-size alphabet:

[1] We use the notation $\log_b^c n = (\log n / \log b)^c$ to denote the c-th power of the base-b logarithm of n. Unless specified, we use $b = 2$.

[2] Zobel et al. [32] and Crauser and Ferragina [3] both mentioned the importance of construction algorithms to the usefulness of the index.

Table 1: Construction times for full-text indices.

index	algorithm	time	space (bits)
SA, CSA, or FM	opt time [19]	$O(n)$	$O(n \log n)$
	opt space [18]	$O(n \log n)$	$O(n)$
	this paper	$O(n)$	$O(n)$
ST or CST	opt time [4]	$O(n)$	$O(n \log n)$
	opt space [12]	$O(n \log n)$	$O(n)$
	this paper	$O(n \log^{\epsilon} n)$	$O(n)$

The acronym ST, SA, CST, CSA, FM represent suffix tree, suffix array, compressed suffix tree, compressed suffix array, and FM-index, respectively. Also, ϵ is any fixed real number with $0 < \epsilon < 1$.

1. An algorithm which constructs the suffix array in $O(n)$ time and $O(n)$-bit working space;

2. An algorithm which constructs the suffix tree in $O(n \log^{\epsilon} n)$ time and $O(n)$-bit working space for $0 < \epsilon < 1$.

To the best of our knowledge, these are the first known algorithms which run in $o(n \log n)$ time and $o(n \log n)$-bit working space.

Besides, our algorithms can actually be adopted to build other full-text indices, including CSA, Compressed Suffix Tree (CST) [12], and FM-index [5]. The performance of our algorithms for constructing these indices are summarized in Table 1. Another application of our algorithm is that, it can act as a time and space efficient algorithm for the block sorting [2], which is a widely used process in various compression schemes, such as `bzip2` [27].

We also study the general case where the alphabet size is not constant. Let Σ be the alphabet, and $|\Sigma|$ denote its size. Our algorithm can construct the suffix array and the suffix tree using $O(n \log |\Sigma|)$-bit working space, while running in $O(n \log \log |\Sigma|)$ time and $O(n(\log^{\epsilon} n + \log |\Sigma|))$ time, respectively, for any fixed ϵ with $0 < \epsilon < 1$. These are the first algorithms that achieve $o(n \log n)$ time with optimal working space. Moreover, for the special case where $\log |\Sigma| = O((\log \log n)^{1-\epsilon})$, we can apply Pagh's data structure for constant-time *rank* queries [24] to further improve the running time of the suffix array construction to the optimal $O(n)$.

Remark. Very recently, Na and Park [23] proposed another algorithm for the construction of CSA, FM-index, and Burrows-Wheeler transform. The running time is $O(n)$ time, which is independent of the alphabet size. The working space is increased slightly to $O(n \log |\Sigma| \log_{|\Sigma|}^{\alpha} n)$ bits, where $\alpha = \log_3 2$.

1.2 The Main Techniques

To achieve small working space, we make use of the Ψ function [6] of CSA and the Burrows-Wheeler (BW) text [2] as our tools, where they can act as an implicit representation of any suffix tree generated during the process of construction. Both the Ψ function and the BW text can be stored in $O(n)$-bit space. Moreover, given them, we can construct the suffix tree and the suffix array in $O(n \log^{\epsilon} n)$ time and $O(n)$ time, respectively for $0 < \epsilon < 1$.

Apart from space concern, another reason for using these two data structures as our tools is that we notice the strengths they possess complement nicely the weaknesses of the other: For the Ψ function, it allows efficient pattern query, while it is difficult to update the function in response to the change in the underlying suffix tree; for the plain BW text, it can be easily and quickly updated, though it does not support efficient pattern query. In our construction algorithm, efficient queries and fast updates are frequently required, so that we use both data structures alternately in order to utilize their strengths.

Another finding that leads to the improvement is related to the backward search algorithm, which is used to find a pattern within the text based on the Ψ function. If we apply a known method [26], given the Ψ function for the text, each step of the algorithm requires $O(\log n)$ time in general. This paper presents a novel auxiliary data structure of $O(n)$ bits which supports each backward search step in $O(\log \log |\Sigma|)$ time instead. As our construction algorithm executes the backward search frequently, the overall running time is thus sped up because of this improvement.

Finally, our algorithm borrows the framework of Farach's linear time suffix tree construction algorithm [4] which first constructs two suffix trees, one for odd-position suffixes and one for even-position suffixes, and then merge the two trees together. The difference, however, lies in the actual implementation, as we have carefully avoided to store the suffix pointers explicitly, thus saving $O(n \log n)$ bits in total.

The remaining of this paper is organized as follows. Section 2 is a preliminary section which gives the definitions for the CSA and Burrows-Wheeler text, and discusses the relationship between them. Section 3 shows the improved result in the backward search algorithm. Section 4 describes the framework for the construction algorithm, while Sections 5 and 6 detail the main steps of the algorithm. In Section 7, we give details to the improvement we can achieve when the alphabet size is sufficiently small, precisely, when $\log |\Sigma| = O((\log \log n)^{1-\epsilon})$. Finally, we give a conclusion in Section 8.

2 Preliminaries

This section is divided into three parts. The first part gives the basic notation, and introduces the suffix array, the Ψ function, and the Burrows-Wheeler text. In the second part, we describe the representation of the Ψ function that is used throughout the paper. Finally, we discuss the duality between the Ψ function and the Burrows-Wheeler text.

2.1 Basic Notation

Firstly, we review some of the basic notation and assumptions. For a text T of length n over an alphabet Σ, it is denoted by $T[0...n-1]$. Each character of Σ is uniquely encoded by an integer in $[0, |\Sigma| - 1]$ which occupies $\log |\Sigma|$ bits. In addition, a character c is alphabetically larger than a character c' if and only if the encoding of c is larger than the encoding of c'. Also, we assume that $T[n-1]$ is a special character that does not appear elsewhere in the text.

For any $i = 0, \ldots, n-1$, the string $T[i..n-1]$ is called a suffix of T. The suffix array $\mathrm{SA}[0..n-1]$ of T is an array of integers such that $T[\mathrm{SA}[i]..n-1]$ is lexicographically the i-th smallest suffix of T. The Ψ_T function, or simply Ψ, is the main component of the CSA [6]. It is defined as follows:

- $\Psi[i] = \mathrm{SA}^{-1}[\mathrm{SA}[i]+1]$ if $\mathrm{SA}[i] \neq n-1$;
- $\Psi[i] = \mathrm{SA}^{-1}[0]$ otherwise.

Immediately, we have the following observation.

i	$W[i]$	SA$[i]$	$\Psi[i]$	$S[\text{SA}[i]...n-1]$
0	g	7	2	$
1	c	2	3	a a c c g $
2	$	0	4	a c a a c c g $
3	a	3	5	a c c g $
4	a	1	1	c a a c c g $
5	a	4	6	c c g $
6	c	5	7	c g $
7	c	6	0	g $

Figure 1: Suffix array, the Ψ function, and the Burrows-Wheeler text W of a string $S = \texttt{acaaccg\$}$.

Observation 1 *For the suffix array and the Ψ function of the text T,*

- *Characters $T[\text{SA}[i]]$ ($i = 1, 2, \ldots, n$) are alphabetically sorted.*
- *Suppose that $T[\text{SA}[i]] = T[\text{SA}[j]]$. Then, $\Psi[i] < \Psi[j]$ if and only if $i < j$.*

On the other hand, the Burrows-Wheeler text W [2] is a transformation on T such that $W[i] = T[\text{SA}[i] - 1]$ if $\text{SA}[i] > 0$, and $W[i] = T[n-1]$ otherwise. Intuitively, $W[i]$ is the preceding character of the i-th smallest suffix of T in T. This transformation process is widely used in various compression schemes such as `bzip2` [27], and it constitutes the main part of the construction of the FM-index [5].

See Figure 1 for an example of the suffix array, the Ψ function, and the Burrows-Wheeler text of a string $S = \texttt{acaaccg\$}$. In the example, we have $\Sigma = \{\texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}\}$ and the last character of S is \$, which is unique among the other characters in S. We also assume that \$ is alphabetically smaller than each character in Σ.

2.2 Representation of Ψ

Observation 1 implies that the Ψ function is piece-wise increasing. In addition, each Ψ value is less than n. Therefore, we can make use of a function $\rho(c, x) = enc(c) \cdot n + x$ and obtain a total increasing function $\Psi'[i] = \rho(T[SA[i]], \Psi[i])$, where $enc(c)$ denotes the encoding of the character c. Note that value of Ψ' is less than $n|\Sigma|$.

Based on the total increasing property, Ψ' can be stored as follows. We divide each $\Psi'[i]$, which takes $\log n + \log|\Sigma|$ bits, into two parts q_i and r_i, where q_i is the first (or most significant) $\log n$ bits, and r_i is the remaining $\log|\Sigma|$ bits. We encode the values $q_0, q_1 - q_0, \ldots, q_{n-1} - q_{n-2}$ in a bit-vector B_1 using unary codes. (Recall that the unary code for an integer $x \geq 0$ is encoded as x `0`'s followed by a `1`.) Note that the encoding has exactly n `1`'s where the $(i+1)$-th `1`, which corresponds to $\Psi[i]$, is at position $i + q_i$. Also, the total number of `0`'s is q_{n-1}, which is at most n. Thus, B_1 uses $2n$ bits. The r_i's are stored explicitly in an array $B_2[0...n-1]$ where each entry occupies $\log|\Sigma|$ bits. Thus, B_2 occupies $n \log|\Sigma|$ bits. Moreover, an auxiliary data structure of $O(n/\log\log n)$ bits is constructed in $O(n)$ time to enable constant-time *rank* and *select* queries, and thus supporting the retrieval of any q_i in constant time [13, 22]. Then, the total size is $n(\log|\Sigma| + 2) + o(n)$ bits. Since q_i and r_i can be retrieved in constant time, so can $\Psi'[i] = |\Sigma|q_i + r_i$. This gives the following lemma.

Lemma 1 *The Ψ' function can be encoded in $O(n \log|\Sigma|)$ bits, so that each $\Psi'[i]$ can be retrieved in constant time.*

Corollary 1 *The Ψ function can be encoded as Ψ' in $O(n \log |\Sigma|)$ bits, so that each $\Psi[i]$ can be retrieved in constant time.*

Proof: The retrieval time follows since $\Psi[i] = \Psi'[i] \bmod n$. □

2.3 Duality between Ψ and W

It is known that Ψ and W are one-to-one corresponding. In the section, we show that the transformation between them can be done in linear time and in $O(n \log |\Sigma|)$-bit space.

We first give a property relating W and Ψ.

Definition 1 *Given an array of characters $x[0...n-1]$, we define the* stable sorting order *of $x[i]$ in x to be the number of characters in x which is alphabetically smaller than $x[i]$, plus the number of characters $x[j]$ with $j < i$ which is equal to $x[i]$. This is in fact the position of $x[i]$ after stable sorting.*

Lemma 2 ([2]) *Let k be the stable sorting order of $W[i]$ in W. Then, $\Psi[k] = i$.*

Proof: Let Y_i denote the suffix $T_{\mathrm{SA}[i]-1}$ when $\mathrm{SA}[i] > 0$, and the suffix $T[n-1]$ otherwise. Note that when $i < j$, if Y_i and Y_j are starting with the same character, Y_i will be lexicographically smaller than Y_j. The reason is that, by excluding the first character, the remaining part of Y_i (which is $T_{\mathrm{SA}[i]}$) is lexicographically smaller than the remaining part of Y_j (which is $T_{\mathrm{SA}[j]}$). Also, observe that the first character of Y_i is equal to $W[i]$. Then, it follows that the stable sorting order of $W[i]$ in W is equal to the rank of Y_i among the set of all Y_i's, which is the set of all suffixes of T.

Thus, we have $k = \mathrm{SA}^{-1}[\mathrm{SA}[i] - 1]$ when $\mathrm{SA}[i] > 0$, and $k = \mathrm{SA}^{-1}[n-1]$ otherwise. In the former case, $\mathrm{SA}[k] = \mathrm{SA}[i] - 1 < n - 1$, so $\Psi[k] = \mathrm{SA}^{-1}[\mathrm{SA}[k] + 1] = \mathrm{SA}^{-1}[\mathrm{SA}[i]] = i$. For the latter case, we have $i = \mathrm{SA}^{-1}[0]$ and $\mathrm{SA}[k] = n - 1$. Thus we have $\Psi[k] = \mathrm{SA}^{-1}[0] = i$. In summary, $\Psi[k] = i$ for all cases, and the lemma follows. □

The next two lemmas show the linear time conversion between W and Ψ.

Lemma 3 *Given W, we can store Ψ in $O(n)$ time and in $O(n \log |\Sigma|)$ bits. The working space is $O(n \log |\Sigma|)$ bits.*

Proof: The conversion is simply based on counting sort. We present the details below for completeness. We construct Ψ' in Section 2.2 from W as an encoding of Ψ (Corollary 1). To construct Ψ', we create a bit-vector $B_1[0...2n-1]$ and initialize all bits to 0. We also create an array $B_2[0...n-1]$ where each entry occupies $\log |\Sigma|$ bits.

Now, we show how to compute the stable sorting order of $W[i]$ in W, for $i = 0, 1, 2, \ldots$. To do so, we use three auxiliary arrays. The first array is L_1 such that $L_1[c]$ stores the number of occurrences of the character c in W. This array can be initialized by scanning W once. The second array is L_2 such that $L_2[c]$ stores the number of occurrences of a character that is smaller than c in W. This array can be initialized by scanning L_1 once. Finally, the third array is L_3 such that $L_3[c]$ stores the number of occurrences of c seen so far. Initially, all entries of L_3 are initialized to 0.

Now, we proceed to read $W[0]$, $W[1]$, and so on. Note that during the process, when we read a character c, we maintain the correctness of L_3 by incrementing $L_3[c]$ just before the next character is read. Thus, at the beginning of

```
1. Compute $L_1[c]$ to store the number of occurrences of each $c \in \Sigma$.
2. Compute $L_2[c]$ to store the number of occurrences of a character that is smaller than $c$.
   That is, $L_2[c] = \sum_{d<c} L_1[c]$ for $c \in \Sigma$.
3. Let $L_3$ be an array such that $L_3[c]$ stores the number of occurrences of the character $c$
   seen so far.
4. Initialize all entries of $L_3$ to be 0.
5. For $i = 1, 2, \ldots, n$
       Let $c = W[i]$, $k = L_2[c] + L_3[c]$
       Compute $x = \rho(c, i)$
       Let $q = x \text{ div } |\Sigma|$, $r = x \bmod |\Sigma|$
       Set $B_1[k+q] = 1$ and $B_2[k] = r$
       Increment $L_3[c]$ by one
6. Compute the $O(n/\log\log n)$-bit auxiliary data structure for $B_1$.
```

Figure 2: Computing Ψ from W.

step i, the counter $L_3[W[i]]$ will be storing the number of occurrences of $W[i]$ in $W[0...i-1]$, and the stable sorting order of $W[i]$ can be computed at by $L_2[W[i]] + L_3[W[i]]$.

Let k be the stable sorting order of $W[i]$ that is computed at step i in the above algorithm. By Lemma 2, $\Psi[k] = i$. Thus, we have $\Psi'[k]$ equals $x = \rho(T[\text{SA}[k]], \Psi[k]) = \rho(W[i], i)$. By our scheme, x is divided into two parts q and r, where $q = x \text{ div } |\Sigma|$ is the first $\log n$ bits, and $r = x \bmod |\Sigma|$ is the remaining bits. For q, a 1 is stored at $B_1[k+q]$. For r, it is stored at $B_2[k]$.

As the stable sorting order of each $W[i]$ is different, all possible $\Psi[k]$ will be computed and stored eventually. A summary of the overall algorithm is shown in Figure 2. It is easy to see that the overall time is $O(n + |\Sigma|)$. For the space complexity, note that L_1, L_2 and L_3 each occupies $|\Sigma| \log n$ bits, which is at most $n \log |\Sigma|$ bits because $|\Sigma| \leq n$. Thus, we use $O(n \log |\Sigma|)$-bit working space. □

Lemma 4 *Given Ψ and T, we can store W in $O(n)$ time and in $O(n \log |\Sigma|)$ bits. The working space is $O(n \log |\Sigma|)$ bits.*

Proof: Let $t = \text{SA}^{-1}[0]$. Then, we have $\Psi[t] = \text{SA}^{-1}[1]$. In general, $\Psi^k[t] = \text{SA}^{-1}[k]$.

Hence, we have $W[\Psi^k[t]] = T[k-1]$. To construct W, we can first compute t. Recall that $T[n-1]$ is a unique character in T. By scanning T, we compare each character of T with $T[n-1]$. Then, we obtain the value $x = \text{SA}^{-1}[n-1]$, which is equal to the number of occurrences of a character in T that is smaller than $T[n-1]$. Then, by definition, $\Psi[x] = \text{SA}^{-1}[0]$, which is equal to t. Thus, t can be found in $O(n)$ time. Afterwards, we iteratively compute $\Psi^i[t]$ and set $W[\Psi^i[t]] = T[i-1]$, for $i = 1$ to n. As $\Psi^i[t]$ corresponds to the rank of a different suffix of T

for different i, all the characters of W will eventually be computed and stored by the above algorithm. The total time of the algorithm is $O(n)$, and the space for W,T, and Ψ are all $O(n \log |\Sigma|)$ bits. The lemma thus follows. □

3 Improving the Backward Search Algorithm

Let S be a text of length m over an alphabet Δ. In this section, we present an $O(m + |\Delta|)$-bit auxiliary data structure for the the Ψ function of S that improves each step in the backward search algorithm from $O(\log m)$ time to $O(\log \log |\Delta|)$ time.

We first present a data structure that supports fast *rank* query in Section 3.1, and show that such a data structure can be constructed efficiently in terms of time and working space. Then, in Section 3.2, we describe the improved backward search algorithm based on the result of Section 3.1.

3.1 Efficient Data Structure for Fast Rank Query

Let Q be a set of distinct numbers. For any integer x, the *rank* of x in Q is the number of elements in Q smaller than x. We begin with two supporting lemmas prior to the description of our data structure. The first one is on perfect hash function, which is obtained by rephrasing the result of Section 4 of [10] as follows.

Lemma 5 *Given x b-bit numbers, where $b = \Theta(\log x)$, a data structure of size $O(xb)$ bits supporting $O(1)$-time existential query can be constructed in $O(x \log x)$ time and $O(xb)$-bit working space.*

The second one is derived from a result in [21, 31] based on Lemma 5.

Lemma 6 *Given z w-bit numbers, where $w = \Theta(\log z)$, a data structure of size $O(zw^2)$ bits supporting $O(\log w)$-time* rank *queries can be constructed in $O(zw \log(zw))$ time and $O(zw^2)$-bit working space.*

Proof: It is shown that *rank* queries can be solved in $O(\log w)$ time, if existential query for all prefixes of the z numbers can be answered in $O(1)$ time [21, 31].[3] The idea is that, given a w-bit number k, its longest common prefix with the z numbers can be found by binary search (on the length) using $O(\log w)$ existential queries, and such a prefix uniquely determines the *rank* of k.

Notice that only $O(zw)$ strings can be a prefix of the z numbers and each can be represented in $\Theta(w)$ bits. Applying Lemma 5 on this set of strings (with $x = O(zw)$ and $b = \Theta(w)$), we have the required data structure. The lemma thus follows. □

Now, we are ready to describe our new data structure, whose performance is summarized below:

[3] In the original papers, the results are for another query called *predecessor*, which finds the largest element in the z numbers that is smaller than the input w-bit number k. However, such a result can be modified easily for the *rank* query as follows. For each number i in the z numbers, it is replaced by the number $iz +$ the rank of i (so that the number now has $w + \log z$ bits), and we construct the *predecessor* data structure for these modified numbers. For the intended *rank* query, we first try to find the predecessor for kz in the modified numbers, and if no predecessor is found, the rank of k in the z numbers is 0. Otherwise, let this predecessor be p. It is easy to see that the required result is equal to $(p \bmod z) + 1$.

Theorem 1 *Let Q be a set of n numbers, each of length $\Theta(\log n)$ bits. Then, a data structure of size $O(n \log n)$ bits supporting $O(\log \log n)$-time* rank *query in Q can be constructed in $O(n)$ time and $O(n \log n)$-bit working space.*

Proof: We construct the following data structure for Q:

1. Let $k_0 < k_1 < \cdots < k_{n-1}$ be n numbers of Q stored in ascending order by an array.
2. Partition the n numbers into n/w^2 lists, each containing w^2 numbers. Precisely, the lists are in the form $\{k_i, k_{i+1}, \ldots, k_{i+w^2-1}\}$, where $i \equiv 0 \pmod{w^2}$.
3. Let the smallest element in each list be its representative. Construct a data structure for *rank* query for these representatives based on Lemma 6.

The above data structure occupies $O(nw)$ bits, and can be constructed in $O(n)$ time and $O(nw)$-bit working space. With such a data structure, the *rank* of x among the n numbers can be found in $O(\log w)$ time as follows.

1. Find the *rank* of x among the n/w^2 representatives of the lists. Let this be r.
2. Then, the *rank* of x among the n numbers must now lie in $[rw^2, (r+1)w^2 - 1]$. Binary search on the w^2 elements $\{k_{rw^2}, k_{rw^2+1}, \ldots, k_{(r+1)w^2-1}\}$ to find the *rank* of x.

Both steps thus take $O(\log w)$ time. This completes the proof of Theorem 1. □

Note that in contrast to the existing data structures for the *rank* query [13, 25], our data structure requires either less space for storage (when compared with [13]), or less time in the construction (when compared with [25]); the drawback is a blow-up in query time. Based on Theorem 1, we can use some extra space to achieve a more generalized result, as shown in the following corollary.

Corollary 2 *Let Q' be a set of n values, each of length $\log \ell + \Theta(\log n)$ bits for any ℓ. Then, a data structure of size $O(n \log n + \ell)$ bits supporting $O(\log \log n)$-time* rank *query in Q' can be constructed in $O(n + \ell)$ time and $O(n \log n + \ell)$-bit working space.*

Proof: The idea is to apply Theorem 1 by transforming the set Q' into another set such that each value takes only $\Theta(\log n)$ bits. Firstly, we scan Q' and create a bit-vector $B[0...\ell - 1]$ such that $B[i] = \texttt{1}$ if there is some number in Q' whose first $\log \ell$ bits represents a value i, and $B[i] = \texttt{0}$ otherwise. Afterwards, we construct an auxiliary data structure for B of size $o(\ell)$ bits to support constant-time *rank* and *select* queries [13, 22].

Now, we transform each number in Q' as follows: if the first $\log \ell$ bit of the number represents the value i, these bits are replaced by the binary bit-sequence for the rank of i in B. Note that the rank of i is less than n, as there are only n numbers. Thus, after the transformation, each value takes $\Theta(\log n)$ bits, and in addition, the transformation preserves the ordering among the elements in Q'.

Let the set of the transformed values be Q. We create the data structure of Theorem 1 on Q. To perform a *rank* query for x in Q' (we assume that x has the same length as any number in Q'), we first obtain the first $\log \ell$ bits of x. Suppose that these bits represent the value i_x. Then there are two cases:

(Case 1) If $B[i_x] = \texttt{1}$, we replace the first $\log \ell$ bits of x by the $\log n$-bits that represents the rank of i_x in B, and obtain a new value y. Then, it is easy to see that the rank of x in Q' is equal to the rank of y in Q.

(Case 2) Otherwise, we replace the first $\log \ell$ bits of x by the $\log n$-bits that represents the rank of i_x in B, while setting the remaining $\Theta(\log n)$ bits to zeroes, and obtain a new value z. Then, it is easy to see that the rank of x in Q' is equal to the rank of z in Q.

Finally, for the time and space complexity, B and its auxiliary data structures can be created in $O(\ell)$ time and stored in $\ell + o(\ell)$ bits, while the data structure for *rank* query in Q can be created in $O(n)$ time and stored in $O(n \log n)$ bits (By Theorem 1). The lemma thus follows. □

3.2 The Improved Backward Search Algorithm

Firstly, a *backward search step* is defined as follows.

Definition 2 *For any pattern P, suppose that the rank of P among all suffixes of S is known. A* backward search step *then computes the rank of cP among the suffixes of S for any character $c \in \Delta$.*

Let Ψ' denote the total increasing function such that $\Psi'[i] = \rho(S[\mathrm{SA}[i]], \Psi[i])$ and $\rho(c, x) = enc(c) \cdot m + x$. Then, we have the following lemma.

Lemma 7 *Let r be the rank of P among all suffixes of S. Then, the rank of cP among all suffixes of S is equal to $j \in [0, m]$ such that $\Psi'[j-1] < \rho(c, r) \leq \Psi'[j]$. (As a sentinel, we let $\Psi'[-1] = -1$ and $\Psi'[m] = m|\Delta|$.)*

Proof: It is easy to check that for all $i = 0, 1, \ldots, j-1$, the rank-i suffix of S must either be starting with a character smaller than c, or starting with c but the remaining part is lexicographically smaller than P. Thus, for all $i = 0, 1, \ldots, j-1$, the rank-i suffix of S is lexicographically smaller than cP. On the other hand, for all $i \geq j$, the rank-i suffix of S is lexicographically greater than or equal to cP. Thus, the rank of cP is j. □

Essentially, a backward search step that computes the rank of cP in the above lemma is equivalent to finding the rank of $\rho(c, r)$ in the set of all Ψ' values. Then based on the data structure for *rank* query in Section 3.1 (Corollary 2), we can obtain the main result of this section as follows.

Lemma 8 *Let S be a text of length m over an alphabet Δ. Suppose that the Ψ function of S is given, which is stored as Ψ' using the scheme in Section 2.2. Then, an auxiliary data structure for the Ψ function of S can be constructed in $O(m + |\Delta|)$ time, which supports each backward search step in $O(\log\log|\Delta|)$ time. The space requirement is $O(m + |\Delta|)$ bits.*

Proof: Let V denote the set of all Ψ' values. To prove the lemma, it suffices to show a data structure of $O(m + |\Delta|)$ bits that supports *rank* query for any x in V in $O(\log\log|\Delta|)$ time.

Firstly, recall that in our encoding of Ψ', each value in V is stored in two parts, where the first $\log m$ bits are encoded by unary codes in a bit-vector B_1, and the remaining $\log|\Delta|$ bits are encoded in an array B_2 as it is. In addition, there is an auxiliary data structure supporting constant-time *rank* and *select* queries.

Let G_i be the set of Ψ' values whose first $\log m$ bits represent the value i. Among the sets of G_i's, we are concerned with those sets whose size is greater than $\log|\Delta|$. Let $G_{i_1}, G_{i_2}, \ldots, G_{i_k}$ be such sets, where $i_1 < i_2 < \cdots < i_k$.

Note that the groups $G_{i_1}, G_{i_2}, \ldots, G_{i_k}$ each has size between $\log|\Delta|$ and $|\Delta|$. Now, we combine the groups, from left to right, into super-groups of size $\Theta(|\Delta|)$. More precisely, we start from G_{i_1}, merge it with G_{i_2}, G_{i_3} and so on, until the size exceeds $|\Delta|$. Then, we merge the next unmerged group with its succeeding group and so on, until the size exceeds $|\Delta|$. The process is repeated until all groups are within a super-group. (To ensure that each super-group has size $\Theta(|\Delta|)$, we add a dummy group $G_m = \{m|\Delta|, m|\Delta|+1, \ldots, (m+1)|\Delta|-1\}$ as a sentinel.)

For each super-group $\mathcal{G}$, let $v_0, v_1, \ldots, v_p$ be its $\Theta(|\Delta|)$ elements. Now, we pick every $\log|\Delta|$ elements (i.e., $v_0, v_{\log|\Delta|}, v_{2\log|\Delta|}, \ldots$), subtract each of them by v_0, and make them the representatives of this super-group. Then, we construct the data structure for *rank* query of Corollary 2 over these representatives.

With the above data structure, *rank* query for any x in $\mathcal{G}$ can be supported as follows. We first check if $x \le v_0$. If so, the *rank* of x is 0. Otherwise, we find the *rank* of $x - v_0$ among the representatives, which takes $O(\log\log|\Delta|)$ time. Suppose the rank is r. Then, the rank of x in $\mathcal{G}$ must lie between $r\log|\Delta|$ and $(r+1)\log|\Delta| - 1$, and this can be found by a binary search in the elements $\{v_{r\log|\Delta|}, \ldots, v_{(r+1)\log|\Delta|-1}\}$ which takes $O(\log\log|\Delta|)$ time. In summary, the time required is $O(\log\log|\Delta|)$.

Now, let us complete the whole picture to show how to perform the *rank* query for x in V. Firstly, we extract the first $\log m$ bits of x by dividing it with $|\Delta|$. Let $i' = x$ div $|\Delta|$ be its value. Next, we determine the size of $G_{i'}$, which can be done in constant-time using *rank* and *select* queries on B_1. If the size is 0 (i.e., $G_{i'}$ is empty), the rank of x in V can be computed immediately (precisely, the required rank is equal to the number of `1`'s in $B_1[0...i'-1]$, which can be computed in constant time using B_1 and its auxiliary data structure). If the size is smaller than $\log|\Delta|$, the rank of x can be found by performing a binary search with the elements in $G_{i'}$, which takes $O(\log\log|\Delta|)$ time. Finally, if the size is greater than $\log|\Delta|$, we locate the super-group $\mathcal{G}$ that contains the elements of $G_{i'}$, and retrieve the rank r of its smallest element v_0 in V. Then, the required rank is r plus the rank of x in $\mathcal{G}$. We now claim that locating the super-group and retrieval of r can be done in constant time (to be proved shortly), so that the total time is $O(\log\log|\Delta|)$.

We prove the above claim as follows. To support finding the smallest element in each super-group, and retrieval of its rank in V, we use a bit-vector B_1' of $O(m)$ bits, obtained from B_1 by keeping only those `1`'s whose corresponding Ψ' value is a smallest element in some super-group. Also, we augment B_1' with constant-time *rank* and *select* data structures. Then, the smallest value of the $(i+1)$-th super-group, and its rank in V, can be found by consulting B_1 and B_1' in constant time. In addition, for any G_i (with size greater than $\log|\Delta|$), the rank of its super-group among the other super-groups can be found by consulting B_1' in constant time.

On the other hand, to support locating the *rank* data structure of the super-group, we first analyze the space requirement of these data structures. For a particular super-group $\mathcal{G} = \{v_0, v_1, \ldots, v_p\}$, the data structure is built for $p/\log\Delta = \Theta(\Delta/\log\Delta)$ elements, each of which has value in $[0, v_p - v_0]$, so that the space is $O(v_p - v_0 + \frac{p}{\log|\Delta|} \cdot \log|\Delta|)$ bits (by Corollary 2), which is $O(v_p - v_0)$ bits since $p \le v_p - v_0$. Thus, the total space requirement is $O(m + |\Delta|)$ bits,[4] and we assume that the data structures of the super-groups are stored consecutively according to the rank of its smallest element. Then, we create a bit-vector B_3 whose length is identical to the above data structures, which is used to mark the starting position of each data structure. Also, we augment the bit-vector with

[4] The additional $O(|\Delta|)$ bits are due to the dummy group G_m.

an $o(m + |\Delta|)$-bit auxiliary data structure to support constant-time *rank* and *select* queries. Thereafter, when we want to locate a super-group for G_i, we find its rank r among the other super-groups using B_1', and then this rank-r super-group can be accessed in constant time using B_3.

In summary, our data structure takes a total space of $O(m + |\Delta|)$ bits and supports each backward search step in $O(\log\log|\Delta|)$ time. For the construction, it takes at most $O(m + |\Delta|)$ time. The lemma thus follows. □

4 The Framework of Constructing CSA and FM-index

Recall that $T[0...n-1]$ is a text of length n over an alphabet Σ, and we assume that $T[n-1]$ is a special character that does not appear elsewhere in T. This section describes how to construct the Ψ function and the Burrows-Wheeler text W of T in $O(n\log\log|\Sigma|)$ time. Our idea is based on Farach's framework for linear-time construction of the suffix tree [4], which first constructs the suffix tree for even-position suffixes by recursion, based on which induces the suffix tree for odd-position suffixes, and then merge the two suffix trees to obtain the required one.

For our case, we first assume that the length of T is a multiple of $2^{\lceil\log\log_{|\Sigma|} n\rceil+1}$. (Otherwise, we add enough \$ and a \$' at the end of T, where \$' is a character alphabetically smaller than the other characters in T, and proceed with the algorithm. The Ψ of this modified string can be converted into the Ψ of T in $O(n)$ time.) Let h be $\lceil\log\log_{|\Sigma|} n\rceil$. For $0 \leq k \leq h$, we define T^k to be the string over the alphabet Σ^{2^k}, which is formed by concatenating every 2^k characters in T to make one character. That is, $T^k[i] = T[i \cdot 2^k + 1..(i+1)\cdot 2^k - 1]$, for $1 \leq i \leq n/2^k$. By definition, $T^0 = T$.

In addition, we introduce the following definitions associating with a string. For any string $S[0...m-1]$ with even number of characters, denote S_e and S_o to be the strings of length $m/2$ formed by *merging* every 2 characters in $S[0...m-1]$ and $S[1...m-1]S[0]$, respectively; more precisely, $S_e[i] = S[2i]S[2i+1]$ and $S_o[i] = S[2i+1]S[2i+2]$, where we set $S[m] = S[0]$. Intuitively, the suffixes of S_e and S_o corresponds to the even-position and odd-position suffixes of S, respectively. We have the following observation.

Observation 2 $T_e^i = T^{i+1}$.

Also, note that the last characters of T_o^i and T_e^i are unique among the corresponding string. This makes the results in Sections 2 and 3 applicable for both texts.

Our basic framework is to use a bottom-up approach to construct Ψ of T^i, or Ψ_{T^i}, for $i = \lceil\log\log_{|\Sigma|} n\rceil$ down to 0, thereby obtaining Ψ of T in the end. Precisely,

- For Step $i = \lceil\log\log_{|\Sigma|} n\rceil$, Ψ_{T^i} is constructed by first building the suffix tree for T^i using Farach's algorithm [4], and then converting it back to the Ψ function.
- For the remaining steps, we construct the Ψ_{T^i} based on the $\Psi_{T^{i+1}}$, the latter of which is in fact $\Psi_{T_e^i}$ by Observation 2. We first obtain $\Psi_{T_o^i}$ based on T^i and $\Psi_{T_e^i}$. Afterwards, we merge $\Psi_{T_o^i}$ and $\Psi_{T_e^i}$ to give Ψ_{T^i}. The complete algorithm is shown in Figure 3.

Sections 5 and 6 describe in details how to obtain $\Psi_{T_o^i}$ from $\Psi_{T_e^i}$ and T^i, and how to merge $\Psi_{T_o^i}$ and $\Psi_{T_e^i}$ to obtain Ψ_{T^i}, respectively. This gives the main theorem of this section.

```
1. For $i = \lceil \log \log_{|\Sigma|} n \rceil$
    (a) Construct suffix tree for $T^i$.
    (b) Construct $\Psi_{T^i}$ from the suffix tree.
2. For $i = \lceil \log \log_{|\Sigma|} n \rceil - 1$ to 0
    (a) Construct $\Psi_{T_o^i}$ based on $\Psi_{T_e^i}$. (Note: $\Psi_{T_e^i} = \Psi_{T^{i+1}}$.)
    (b) Construct $\Psi_{T^i}$ based on the $\Psi_{T_o^i}$ and $\Psi_{T_e^i}$.
```

Figure 3: The construction algorithm of Ψ function of T.

Theorem 2 *The Ψ function and the Burrows-Wheeler text W of T can be constructed in $O(n \log \log |\Sigma|)$ time and $O(n \log |\Sigma|)$-bit working space.*

Proof: We refer to the algorithm in Figure 3, which has two phases. For Phase 1, we have $i = \lceil \log \log_{|\Sigma|} n \rceil$. We first construct the suffix tree for T^i whose size is $n/2^{\lceil \log \log_{|\Sigma|} n \rceil} \le n \log |\Sigma| / \log n$. This requires $O(n \log |\Sigma| / \log n)$ time and $O(n \log |\Sigma|)$-bit space by using Farach's suffix tree construction algorithm [4]. Then, Ψ_{T^i} can be constructed in $O(n \log |\Sigma| / \log n)$ time and $O(n \log |\Sigma|)$-bit working space. Thus, Phase 1 in total takes $O(n)$ time and $O(n \log |\Sigma|)$-bit space.

For every Step i in Phase 2, we construct Ψ_{T^i}. Let Δ_i be the alphabet of T^i. Then, Part (a) takes $O(|T^i| + |\Delta_i|)$ time (Lemma 12), and Part (b) takes $O(|T^i| \log \log |\Delta_i| + |\Delta_i|)$ time (Lemma 14). For the space, both require $O(|T^i| \log |\Delta_i| + |\Delta_i|)$ bits. Note that $|T^i| = n/2^i$ and $|\Delta_i| \le |\Sigma|^{2^i} \le n$, so the space used by Step i is $O(|T^i| \log |\Delta_i| + |\Delta_i|) = O(n \log |\Sigma|)$ bits, and the time is $O(|T^i| \log \log |\Delta_i| + |\Delta_i|) = O((n/2^i) \cdot (i + \log \log |\Sigma|) + |\Sigma|^{2^i})$. In total, the space for Phase 2 is $O(n \log |\Sigma|)$ bits and the time is:

$$\begin{aligned} & \sum_{i=1}^{\lceil \log \log_{|\Sigma|} n \rceil - 1} O\left(\frac{n}{2^i} (i + \log \log |\Sigma|) + |\Sigma|^{2^i} \right) \\ = \; & O(n \log \log |\Sigma|). \end{aligned}$$

The whole algorithm for constructing Ψ of T therefore takes $O(n \log \log |\Sigma|)$ time and $O(n \log |\Sigma|)$-bit space. Finally, the Burrows-Wheeler text W can be constructed from Ψ using Lemma 4 in $O(n)$ time and $O(n \log |\Sigma|)$-bit space. This completes the proof. □

Once the Burrows-Wheeler transformation is completed, FM-index can be created by encoding the transformed text W using Move-to-Front encoding and Run-Length encoding [5]. When the alphabet size is small, precisely, when $|\Sigma| \log |\Sigma| = O(\log n)$, Move-to-Front encoding and Run-Length encoding can be done in $O(n)$ time based on a pre-computed table of $o(n)$ bits. In summary, this encoding procedure takes $O(n)$ time using $o(n)$-bit space in addition to the output index. Thus, we have the following result.

		X_i		
i	$y[i]$	$x[i]$	$S_e[SA_e[i]...m/2-1]$	$S[0]$
0	`$a`	`c`	`aa cc g$`	`a`
1	`cg`	`$`	`ac aa cc g$`	`a`
2	`ca`	`a`	`cc g$`	`a`
3	`ac`	`c`	`g$`	`a`

(a)

i	$y \rightarrow C_o$	$x \rightarrow$ sorted x
0	`cg`	`$`
1	`ca`	`a`
2	`$a`	`c`
3	`ac`	`c`

(b)

Figure 4: Consider $S =$ `acaaccg$`. (a) The relationship between $x[i]$, $y[i]$ and X_i. Note that X_i corresponds to a suffix of S_o. (b) After stable sorting on the array x, the array y becomes C_o.

Theorem 3 *The FM-index of T can be constructed in $O(n \log\log |\Sigma|)$ time and $O(n \log |\Sigma|)$-bit working space, when $|\Sigma| \log |\Sigma| = O(\log n)$.*

4.1 Further Discussion

The compressed suffix tree (CST) [12] is a compact representation of the suffix tree taking $O(n \log |\Sigma|)$ bits of space. The core of the CST consists of (1) CSA of the input text, (2) parentheses encoding of the tree structure of the suffix tree, and (3) an *Hgt* array that enables efficient computation of the longest common prefix (LCP) query. It is shown in [11, 12] that once the CSA of the input text is computed, the CST can be constructed in $O(n \log^{\epsilon} n)$ time and $O(n \log |\Sigma|)$-bit working space, for any fixed ϵ with $0 < \epsilon < 1$.

Once the CST is constructed, we can simulate a pre-order traversal of the original suffix tree in $O(n(\log^{\epsilon} n + \log |\Sigma|))$ time [11, 12], thereby constructing the original suffix tree along the traversal. Summarizing, we have the following result:

Theorem 4 *The CST and suffix tree of T can be constructed in $O(n \log^{\epsilon} n)$ time and $O(n(\log^{\epsilon} n + \log |\Sigma|))$ time, respectively, for any fixed ϵ with $0 < \epsilon < 1$. Both construction algorithms require $O(n \log |\Sigma|)$-bit working space.*

5 Constructing Ψ_{S_o}

Given $S[0...m-1]$ and Ψ_{S_e}, this section describes how to construct Ψ_{S_o}. Our approach is indirect, as prior to obtaining Ψ_{S_o}, we need to construct the Burrows-Wheeler text C_o of S_o.

Let Δ be the alphabet of S. Define $x[0..m/2-1]$ to be an array of characters such that $x[i] = S[2SA_e[i]-1]$ where $2SA_e[i]-1$ is computed in modulo-m arithmetic. Let X_i be the string $x[i]S_e[SA_e[i]..m/2-1]S[0]$.

Observation 3 *X_i is a suffix of S_o if $SA_e[i] \neq 0$. Otherwise, the first character of X_i is $S[m-1]$, which is unique among other characters in S.*

Let X be the set $\{X_k | 0 \leq k \leq m/2-1\}$. Intuitively, X is the same as the set of suffixes of S_o. See Figure 4(a) for an example of X_i.

Lemma 9 *The stable sorting order of $x[i]$ in x equals the rank of X_i in X.*

Proof: By omitting the first characters of every X_i's, they are of the form $S_e[SA_e[i]..m/2-1]S[0]$, which are already sorted. Thus, the rank of X_i is equal to the stable sorting order of $x[i]$ in x. □

Lemma 10 *Given Ψ_{S_e} and S, we can construct C_o in $O(m + |\Delta|)$ time and $O(m \log |\Delta| + |\Delta|)$-bit space.*

Proof: Let $y[0..m/2-1]$ be an array such that $y[i]$ stores the two characters that immediately precede $x[i]$ in S (i.e., $S[2SA_e[i]-3]S[2SA_e[i]-2]$). In fact, $y[i]$ is the preceding character of X_i in S_o. Using similar approach as in Lemma 4, x and y can be computed in $O(m)$ time, and both arrays occupy $O(m \log |\Delta|)$ bits.

To construct C_o, we perform a stable sort on x as in Lemma 3, and iteratively compute the stable sorting order k of $x[i]$, which is equal to the rank of X_i by Lemma 9. During the process, we set $C_o[k] = y[i]$. The total time is $O(m + |\Delta|)$ and the total space is $O(m \log |\Delta| + |\Delta|)$ bits. See Figure 4 for an example. □

Lemma 11 *Ψ_{S_o} can be constructed from C_o in $O(m + |\Delta|)$ time and $O(m \log |\Delta| + |\Delta|)$-bit space.*

Proof: The proof is similar to Lemma 3. □

Thus, we conclude this section with the following lemma.

Lemma 12 *Given Ψ_{S_e} and S, we can construct Ψ_{S_o} in $O(m + |\Delta|)$ time and $O(m \log |\Delta| + |\Delta|)$-bit space.*

6 Merging Ψ_{S_o} and Ψ_{S_e}

In this section, we construct Ψ_S from Ψ_{S_o} and Ψ_{S_e}. The idea is to determine the rank of any suffix of S among all suffixes of S, and based on this information, we construct the Burrows-Wheeler text C of S. Finally, we convert C to Ψ_S by Lemma 3.

Let s be any suffix of S. Observe that the rank of s among the suffixes of S, is equivalent to the sum of the rank of s among the odd-position suffixes and that among the even-position suffixes of S. Based on this observation, we can construct the C array (the Burrows-Wheeler transformation of S) as follows.

Firstly, we construct the auxiliary data structures of Lemma 8 for Ψ_{S_o} and for Ψ_{S_e}. Next, we perform backward searches for S_e on Ψ_{S_o} and Ψ_{S_e} simultaneously by Lemma 7, so that at step i, we obtain the ranks of $S_e[m/2 - i...m/2-1]$ among the odd-position suffixes and even-position suffixes of S, respectively. By summing these two ranks, we get the rank k of $S_e[m/2-i...m/2-1]$ among all suffixes of of S. Then, we set $C[k]$ to be $S[m-2i-1]$, which is the preceding character of the suffix $S[m-2i...m-1] = S_e[m/2-i...m/2-1]$.

Similarly, we perform a simultaneous backward search for S_o on Ψ_{S_o} and Ψ_{S_e} to complete the remaining entries of C. Thus, we obtain C by $O(m)$ backward search steps. The algorithm is depicted as MERGECSA in Figure 5.

The following lemma shows the correctness of our algorithm.

Lemma 13 *The algorithm* MERGECSA *in Figure 5 correctly constructs $C[0...m-1]$.*

```
MERGECSA
  1. Construct the auxiliary data structures for $\Psi_{S_o}$ and for $\Psi_{S_e}$ to support efficient backward search.
  2. Backward search for $S_e$ on $\Psi_{S_o}$ and $\Psi_{S_e}$ simultaneously, and
      (a) at Step $i$, we obtain the rank of $S_e[m/2 - i...m/2 - 1]$ among the odd-position suffixes and
          that among the even-position suffixes of $S$. Let the sum of the ranks be $k$.
      (b) Set $C[k] = S[m - 2i - 1]$.
  3. Backward search for $S_o$ on $\Psi_{S_o}$ and $\Psi_{S_e}$ simultaneously, and fill in $C[k]$ accordingly.
```

Figure 5: Merging Ψ_{S_o} and Ψ_{S_e}.

Proof: Recall that for every suffix $S[i..m-1]$, $C[SA^{-1}[i]]$ equals the preceding character of $S[i...m-1]$. For every even-position suffix $S[i...m-1] = S_e[i/2...m/2-1]$, Step 2 computes its rank k among all odd-position and even-position suffixes. By definition, $k = SA^{-1}[i]$. Therefore, Step 2 correctly assigns $C[k]$ to be the preceding character of $S[i..m-1]$. By the same argument, Step 3 handles the odd-position suffixes and correctly assigns $C[SA^{-1}[i]]$ to be the preceding character of $S[i..m-1]$.

Therefore, after Steps 2 and 3, MERGECSA completely constructs $C[0..m-1]$. The lemma thus follows. □

By Lemma 8, the auxiliary data structures can be constructed in $O(m + |\Delta|)$ time and $O(m + |\Delta|)$-bit space, and then each backward search step is done in $O(\log \log |\Delta|)$ time. On the other hand, the Ψ functions occupies $O(m \log |\Delta|)$-bit space. Thus, we have the following lemma.

Lemma 14 *Given Ψ_{S_o} and Ψ_{S_e}, we can construct Ψ_S in $O(m \log \log |\Delta| + |\Delta|)$ time and $O(m \log |\Delta| + |\Delta|)$-bit space.*

7 Improvement when $\log |\Sigma| = O((\log \log n)^{1-\epsilon})$

In case the alphabet size is small, precisely, when $\log |\Sigma| = O((\log \log n)^{1-\epsilon})$, we can improve the construction time of CSA and FM-index to $O(n)$, which is optimal. The improvement is based on the following data structure of Pagh for supporting constant-time *rank* queries [24].

Theorem 5 [24] *Given n distinct numbers in $[0, m-1]$ such that $m = n \log^{O(1)} n$, a data structure of size $B + O(\frac{n(\log \log n)^2}{\log n})$ bits supporting constant-time* rank *queries can be constructed in $O(n)$ time and $O(B)$-bit space where $B = \lceil \log \binom{m}{n} \rceil = n \log \frac{m}{n} + O(n)$.*

We apply the same algorithm as in Section 4 for the construction of CSA, but we make changes only in the encodings of Ψ_{T^i}, for $i < \log \log \log |\Sigma|$. For those values of i, we have $|T^i| = n/2^i$ and the alphabet size of T^i

is $|\Sigma|^{2^i}$. When $\log|\Sigma| = O((\log\log n)^{1-\epsilon})$, we have $|\Sigma|^{2^i} = \log^{O(1)}|T^i|$.[5] Thus, the total increasing sequence of such Ψ'_{T^i}'s can be encoded by Theorem 5, and each backward search step on these Ψ functions can be done in constant time. This gives the following theorem.

Theorem 6 *If $\log|\Sigma| = O((\log\log n)^{1-\epsilon})$, the CSA, FM-index and the Burrows-Wheeler text W of T can be constructed in $O(n)$ time and $O(n\log|\Sigma|)$-bit working space.*

Proof: We refer to the algorithm in Figure 3. After the change in the encodings of Ψ_{T^i} for $i < \log\log\log|\Sigma|$, the time required by each phase is as follows.

- Phase 1 takes $O(n)$ time;
- For $i \geq \log\log\log|\Sigma|$, Step i in Phase 2 takes $O((n/2^i)\cdot(i+\log\log|\Sigma|)+|\Sigma|^{2^i})$ time;
- For $i < \log\log\log|\Sigma|$, Step i in Phase 2 takes $O(n/2^i+|\Sigma|^{2^i})$ time.

It follows that the total time required is $O(n)$. For the space complexity, it remains $O(n\log|\Sigma|)$ bits. Thus, the CSA and the Burrows-Wheeler text W of T can be constructed in the stated time and space, while the FM-index can be constructed in $O(n)$ time and $O(n\log|\Sigma|)$-bit space once W is obtained. This completes the proof. □

8 Concluding Remarks

We have shown that suffix trees, suffix arrays, and other full-text indices can be constructed in $o(n\log n)$ time and $o(n\log n)$-bit space, giving a positive answer to an open problem.

Very recently linear-time algorithms for constructing suffix arrays have been proposed [14–16]. Though using interesting techniques, their algorithms require $O(n\log n)$-bit working space, and they will imply only $O(n\log^\epsilon n)$ time algorithms for constructing suffix arrays if the working space is limited to $O(n\log|\Sigma|)$ bit. Thus, the algorithm proposed in this paper is best-suited for suffix array construction under practical consideration, where the input text is very long but alphabet size is small.

One of the open problem remains is that, whether we can construct suffix tree in optimal $O(n)$ time for texts with general alphabet, while using optimal $O(n\log|\Sigma|)$-bit working space. Another direction of research is to further reduce the working space for constructing full-text indices, from $O(n\log|\Sigma|)$ bits to an input-dependent $O(nH)$ bits where H is the entropy of the input text.

Acknowledgment

The authors would like to thank Roberto Grossi, Tak-Wah Lam, Takeshi Tokuyama, and the anonymous reviewers for helpful comments, and Ankur Gupta for sending us his paper. The work of the first author is supported in part by the Hong Kong RGC Grant HKU-7024/01E while studying at the University of Hong Kong. The work of the second author is supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan. The work of the third author is supported in part by the NUS Academic Research Grant R-252-000-119-112.

[5] This can be seen by considering the boundary case of $i = \log\log\log|\Sigma|$, so that $|T^i|$ becomes smallest while the alphabet size becomes largest.

References

[1] P. Beame and F. E. Fich. Optimal Bounds for the Predecessor Problem and Related Problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002. Preliminary version appears in STOC 1999.

[2] M. Burrows and D. J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, CA, USA, 1994.

[3] A. Crauser and P. Ferragina. A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory. *Algorithmica*, 32:1–35, 2002.

[4] M. Farach. Optimal Suffix Tree Construction with Large Alphabets. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, pages 137–143, 1997.

[5] P. Ferragina and G. Manzini. Indexing Compressed Text. *Journal of the ACM*, 52(4):552–581, 2005. Preliminary version appears in FOCS 2000.

[6] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. Preliminary version appears in STOC 2000.

[7] D. A. Grossman and O. Frieder. *Information Retrieval: Algorithms and Heuristics*. Kluwer Academic Publishers, Boston, MA, USA, 1998.

[8] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA, 1997.

[9] T. Hagerup. Sorting and Searching on the Word RAM. In *Proceedings of Symposium on Theory Aspects of Computer Science*, pages 366–398, 1998.

[10] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic Dictionaries. *Journal on Algorithms*, 41(1):69–85, 2001.

[11] W. K. Hon. *On the Construction and Application of Compressed Text Indexes*, Ph.D. Thesis, University of Hong Kong, 2004.

[12] W. K. Hon and K. Sadakane. Space-Economical Algorithms for Finding Maximal Unique Matches. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 144–152, 2002.

[13] G. Jacobson. Space-Efficient Static Trees and Graphs. In *Proceedings of Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[14] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear Work Suffix Array Construction. *Journal of the ACM*, 53(6):918–936, 2006. Preliminary version appears in ICALP 2003.

[15] D. Kim, J. Sim, H. Park, and K. Park. Constructing Suffix Arrays in Linear Time. *Journal of Discrete Algorithms*, 3(2–4):126–142, 2005. Preliminary version appears in CPM 2003.

[16] P. Ko and S. Aluru. Space Efficient Linear Time Construction of Suffix Arrays. *Journal of Discrete Algorithms*, 3(2–4):143–156, 2005. Preliminary version appears in CPM 2003.

[17] S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software Practice and Experiences*, 29:1149–1171, 1999.

[18] T. W. Lam, K. Sadakane, W. K. Sung, and S. M. Yiu. A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays. In *Proceedings of International Conference on Computing and Combinatorics*, pages 401–410, 2002.

[19] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[20] E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[21] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, Heidelberg, Germany, 1984.

[22] J. I. Munro. Tables. In *Proceedings of Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42, 1996.

[23] J. C. Na and K. Park. Alphabet-Independent Linear-Time Construction of Compressed Suffix Arrays Using $o(n \log n)$-bit Working Space. *Theoretical Computer Science*, 385(1–3):127–136, 2007. Preliminary version appears in CPM 2005.

[24] R. Pagh. Low Redundancy in Static Dictionaries with Constant Query Time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[25] R. Raman, V. Raman, and S. S. Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees and Multisets. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, 2002.

[26] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003. Preliminary version appears in ISAAC 2000.

[27] J. Seward. The `bzip2` and `libbzip2` official home page, 1996. `http://www.bzip.org/`.

[28] S. Shimozono, H. Arimura, and S. Arikawa. Efficient Discovery of Optimal Word Association Patterns in Large Text Databases. *New Generation Computing*, 18:49–60, 2000.

[29] E. Ukkonen. On-line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, 1995.

[30] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[31] D. E. Willard. Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

[32] J. Zobel, A. Moffat, and K. Ramamohanarao. Guidelines for Presentation and Comparison of Indexing Techniques. *SIGMOD Record*, 25(3):10–15, 1996.